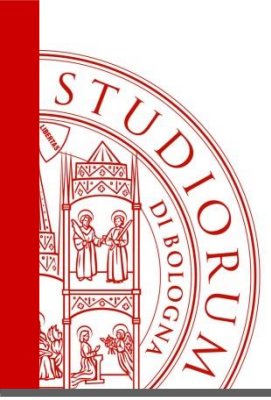


Università degli Studi di Bologna
Corso di Laurea in Ingegneria Informatica

Version Control Systems

Ingegneria del Software T

Prof. MARCO PATELLA
Dipartimento di Informatica – Scienza e Ingegneria (DISI)

Version Control

The management of changes to:

- documents
- computer programs
- large web sites
- and other collections of information

Other names:

- Revision control
- Source control
- Source code management

(source: wikipedia)

Dealing with change

The need for a logical way to organize and control revisions has existed for almost as long as writing has existed, but revision control became much more important, and complicated when the era of computing began

(source: wikipedia)

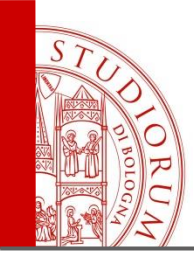
Scenarios of use:

- (Source) code
- Technical drawing
- Textual documentation
- Laws
- … (virtually anything that can be changed)

The case of software

All software has multiple versions

- Different releases of a product
- Variations for different platforms
  - Hardware and software
- Versions within a development cycle
  - Test release with debugging code
  - Alpha, beta of final release
- Each time you edit a program

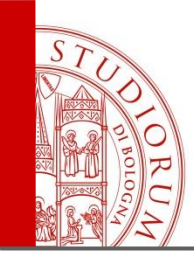


Version control

Version control tracks multiple versions

In particular, allows

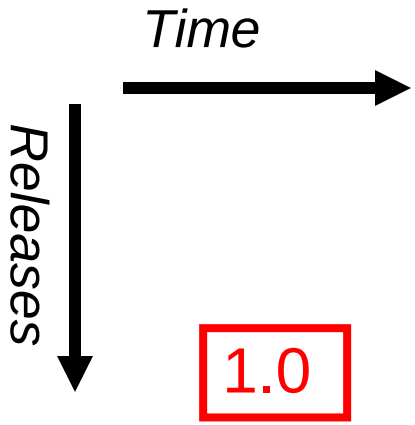
- old versions to be recovered
- multiple versions to exist simultaneously



Why Use Version Control?

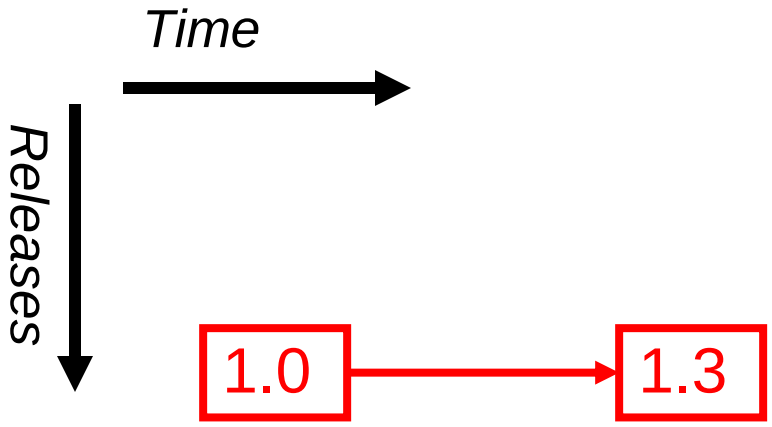
- Because everyone does
  - A basic software development tool
- Because it is useful
  - You will want old/multiple versions
  - Ability to recreate project history
  - Backup, anyone?

Scenario I: Bug Fix



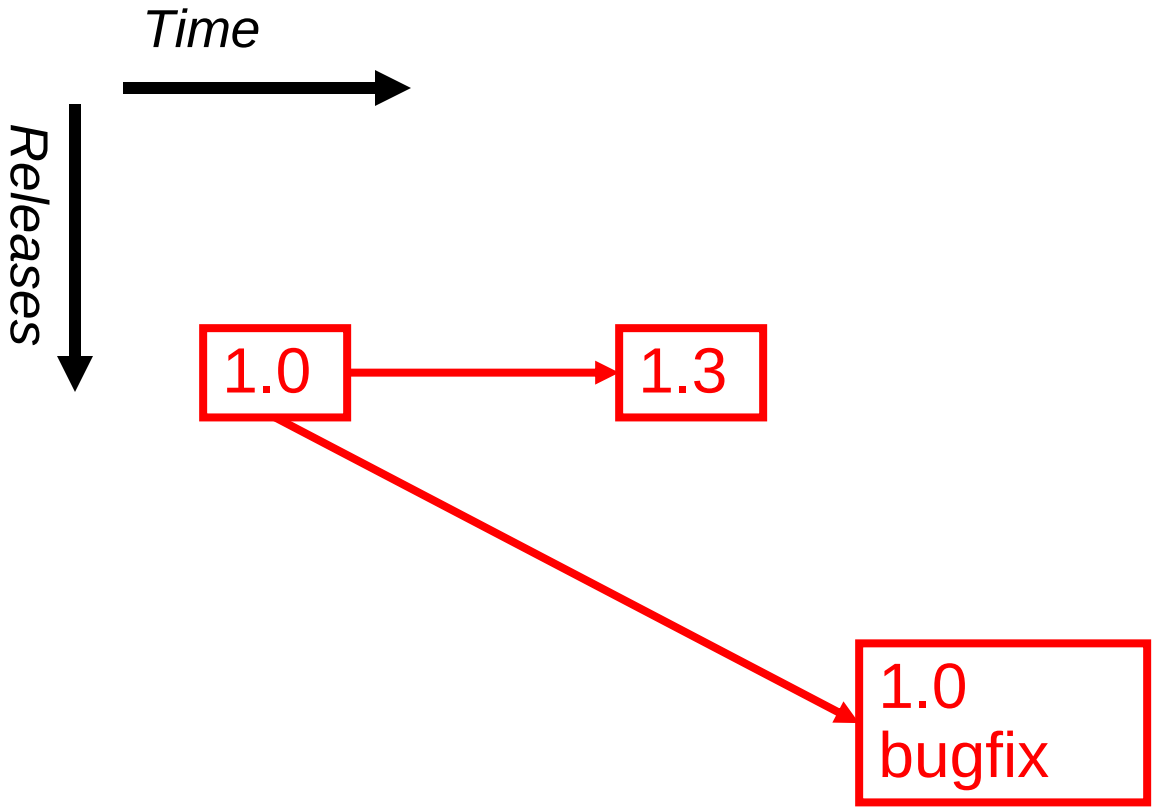
First public release of the hot new product

Scenario I: Bug Fix



Internal development continues, progressing to version 1.3

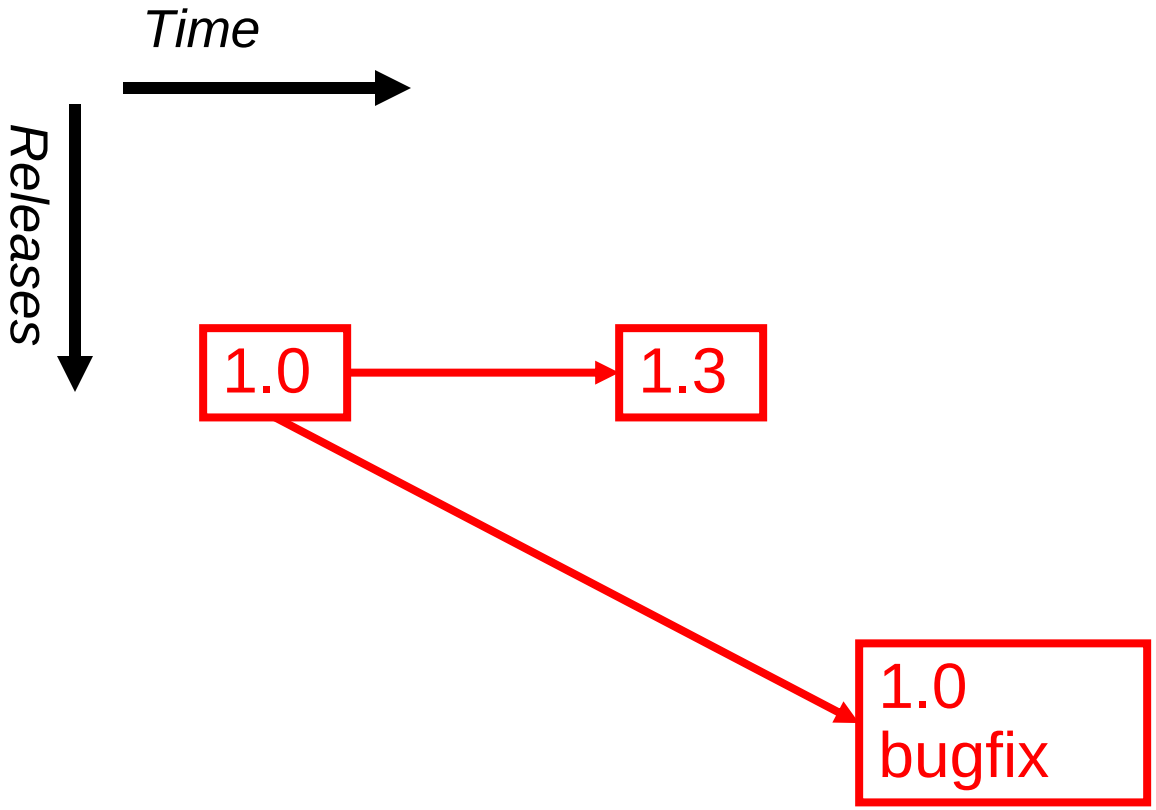
Scenario I: Bug Fix



A fatal bug is discovered in the product (1.0), but 1.3 is not stable enough to release.

Solution: Create a version based on 1.0 with the bug fix.

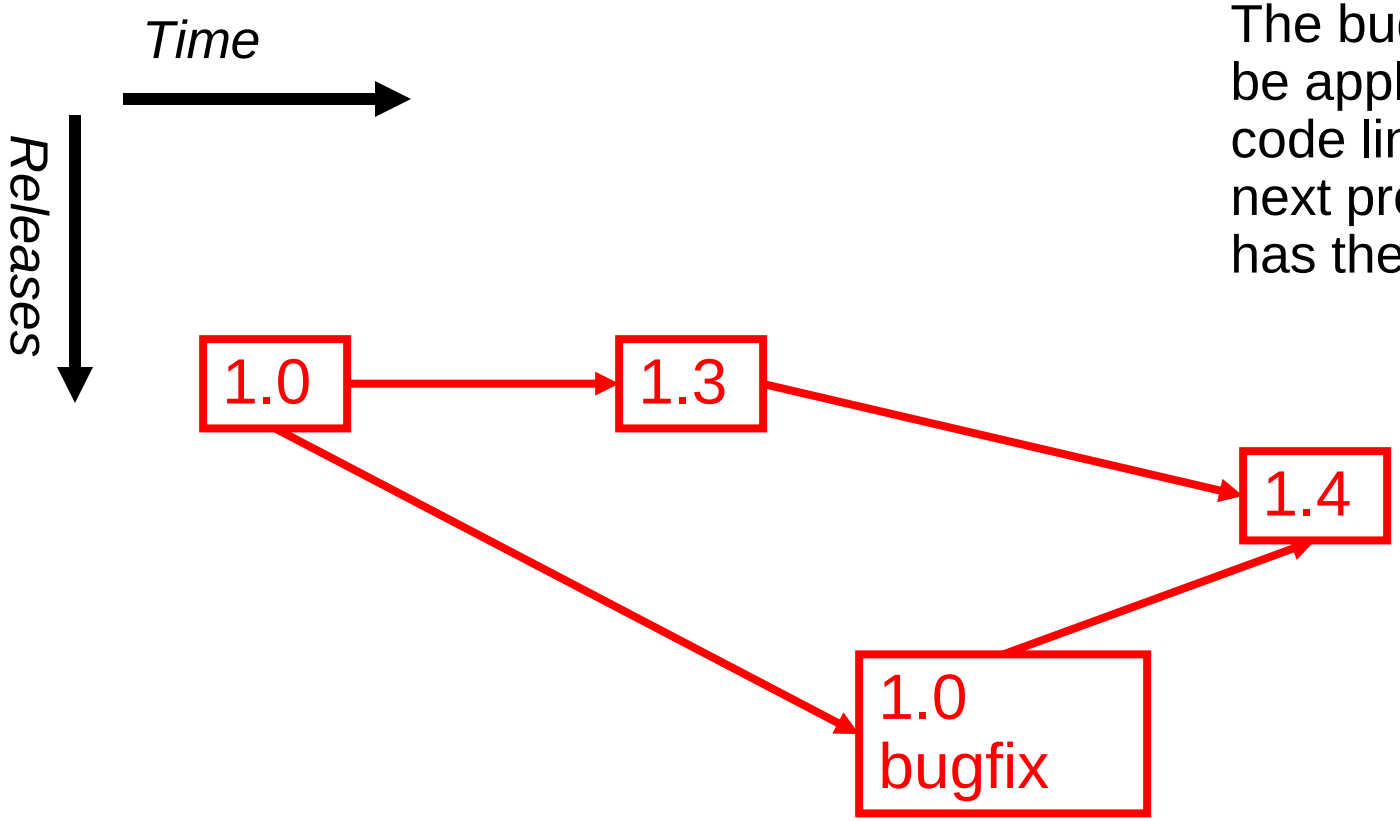
Scenario I: Bug Fix



Note that there are now two lines of development beginning at 1.0.

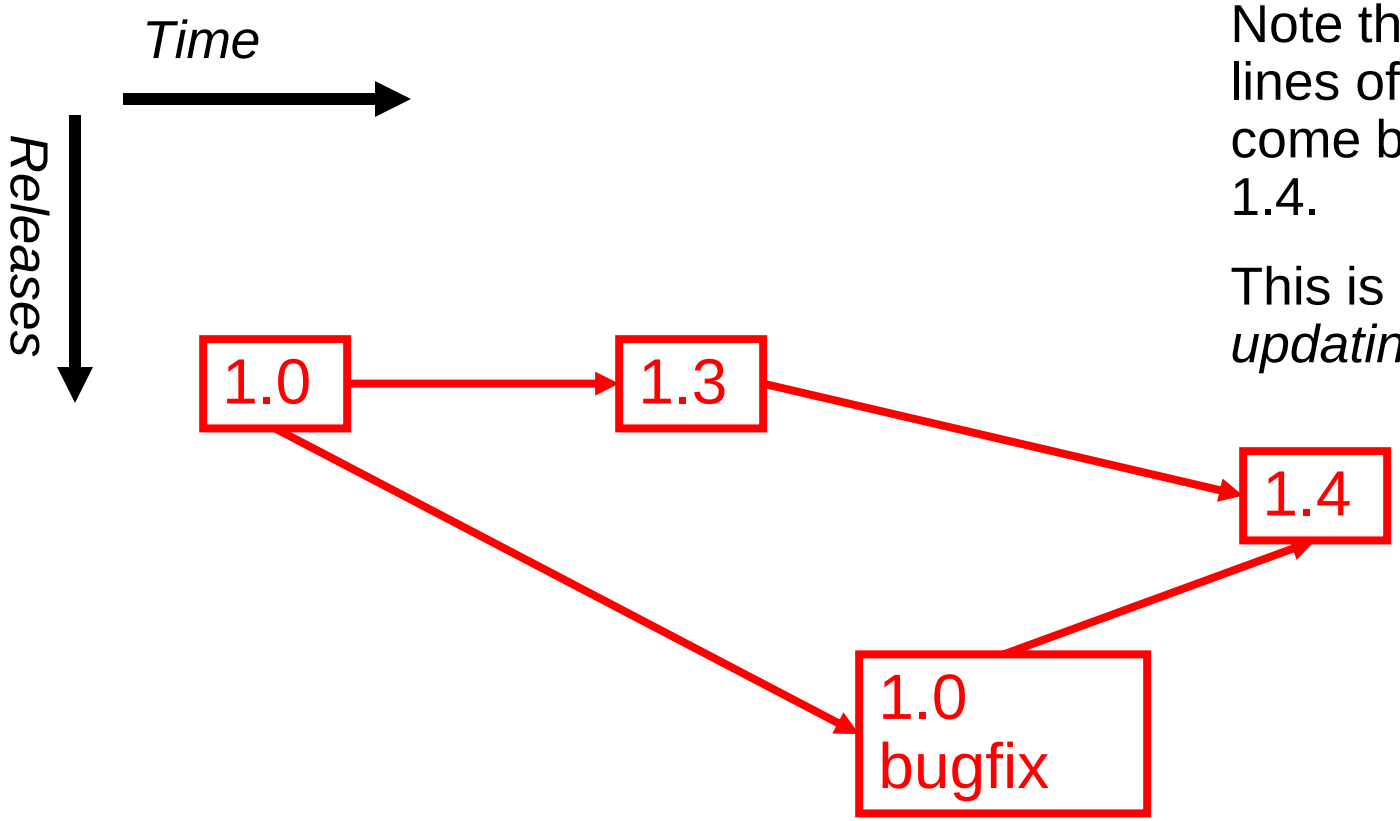
This is *branching.*

Scenario I: Bug Fix



The bug fix should also be applied to the main code line so that the next product release has the fix.

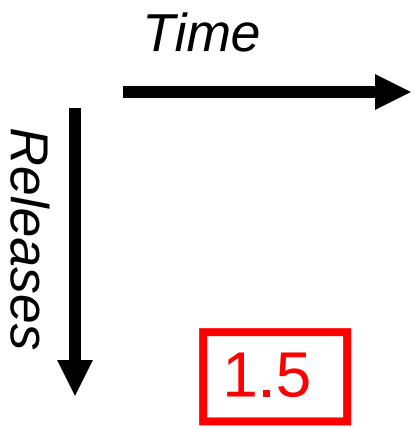
Scenario I: Bug Fix



Note that two separate lines of development come back together in 1.4.

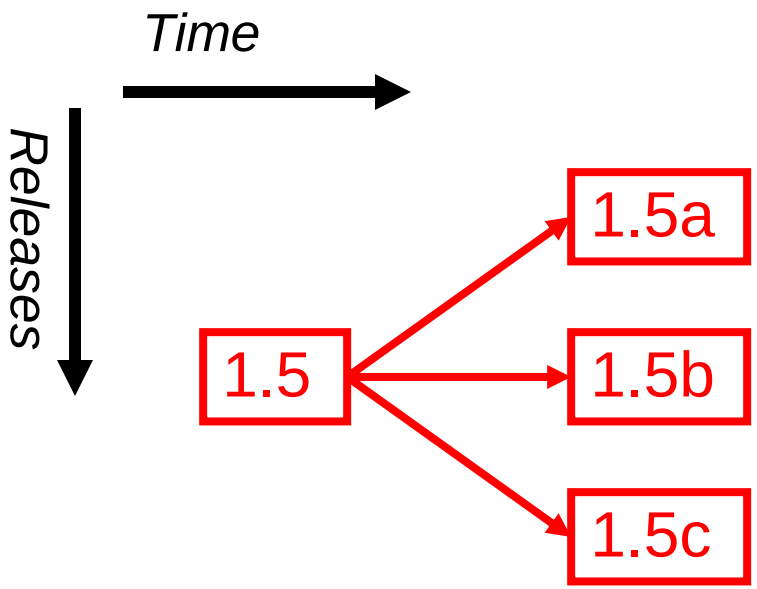
This is *merging* or *updating*.

Scenario II: Normal Development



You are in the middle of a project with three developers named a, b, and c.

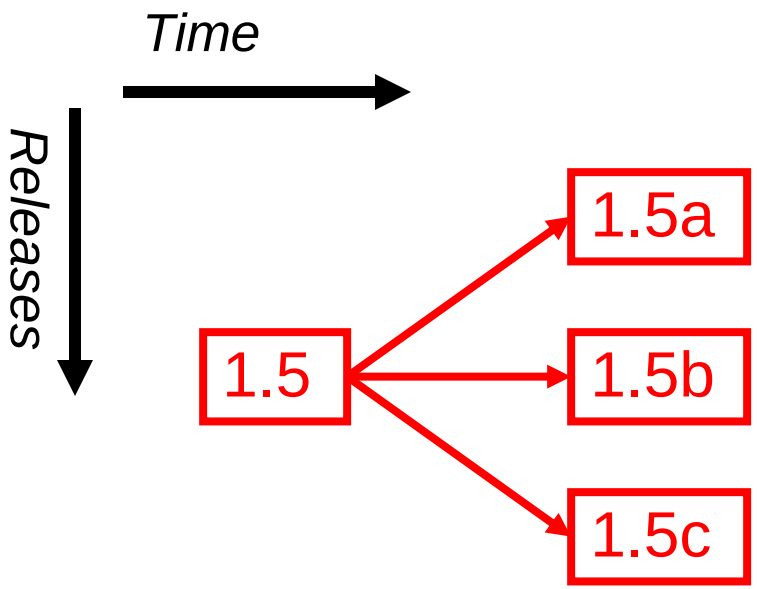
Scenario II: Normal Development



At the beginning of the day everyone *checks out* a copy of the code.

A check out is a local working copy of a project, outside of the version control system. Logically it is a (special kind of) branch.

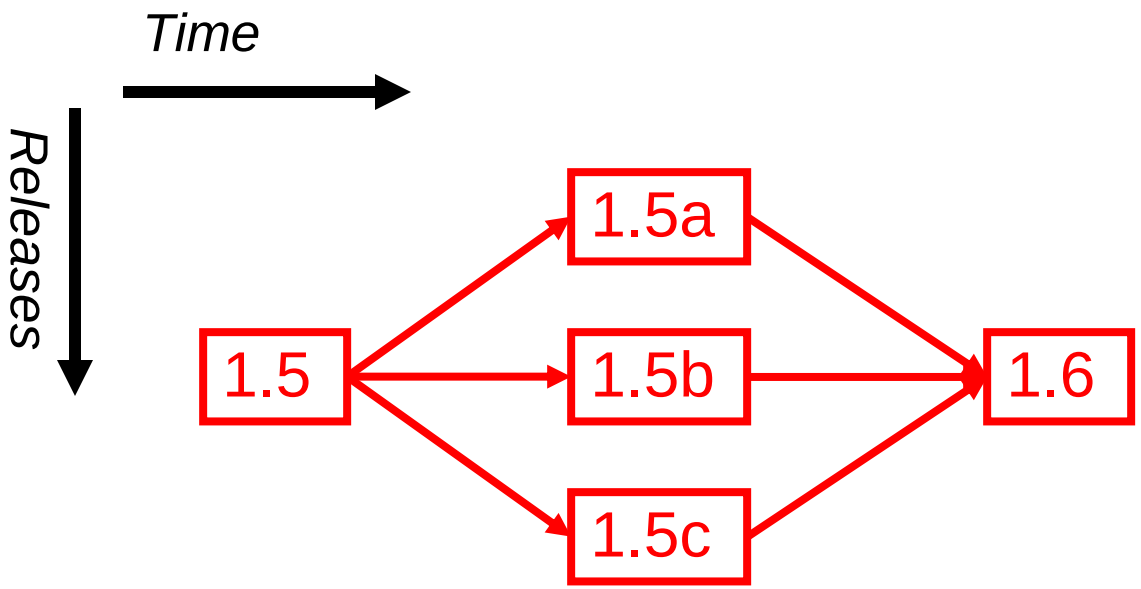
Scenario II: Normal Development



The local versions isolate the developers from each other's possibly unstable changes.

Each builds on 1.5, the most recent stable version.

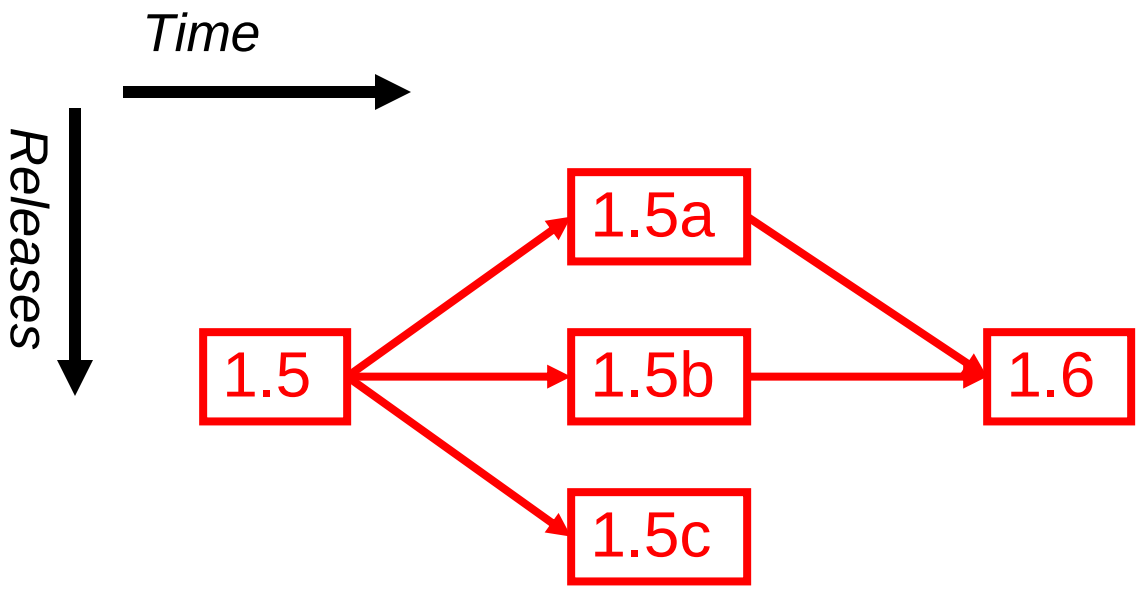
Scenario II: Normal Development



At 4:00 pm everyone *checks in* their tested modifications.

A check in is a kind of merge where local versions are copied back into the version control system.

Scenario II: Normal Development

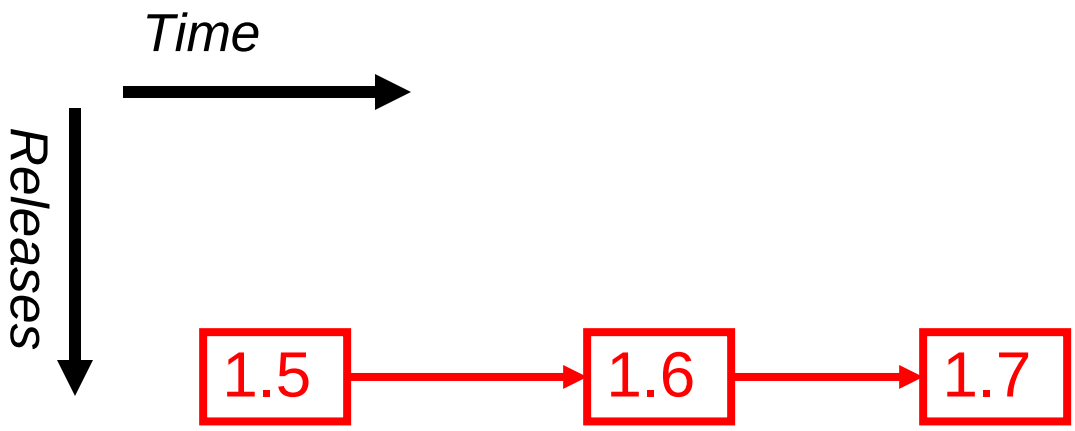


In many organizations check in automatically runs a test suite against the result of the check in.

If the tests fail the changes are not accepted.

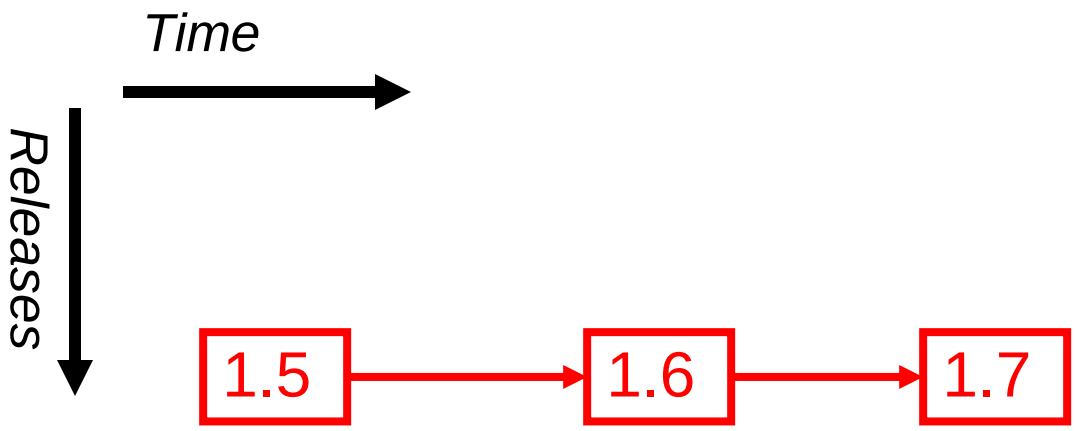
This prevents a sloppy developer from causing all work to stop by, e.g., creating a version of the system that does not compile.

Scenario III: Debugging



You develop a software system through several revisions.

Scenario III: Debugging

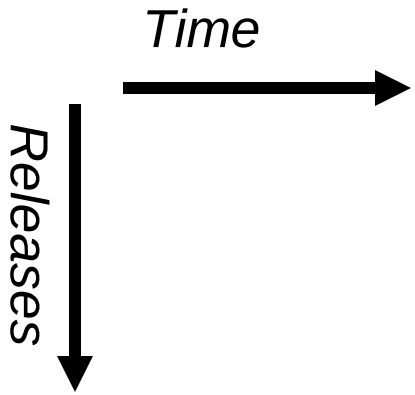


In 1.7 you suddenly discover a bug has crept into the system.

When was it introduced?

With version control you can check out old versions of the system and see which revision introduced the bug.

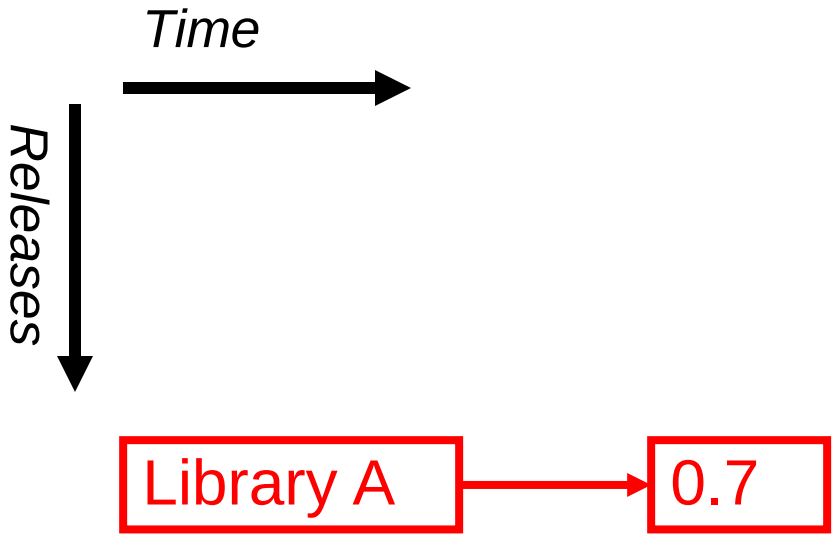
Scenario IV: Libraries



Library A

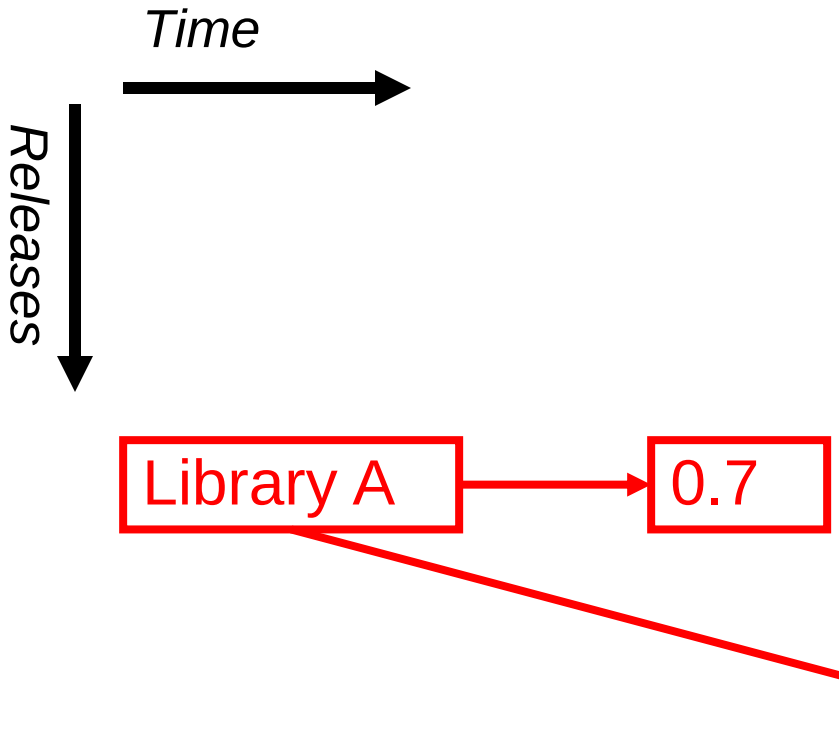
You are building software on top of a third-party library, for which you have source.

Scenario IV: Libraries



You begin implementation of your software, including modifications to the library.

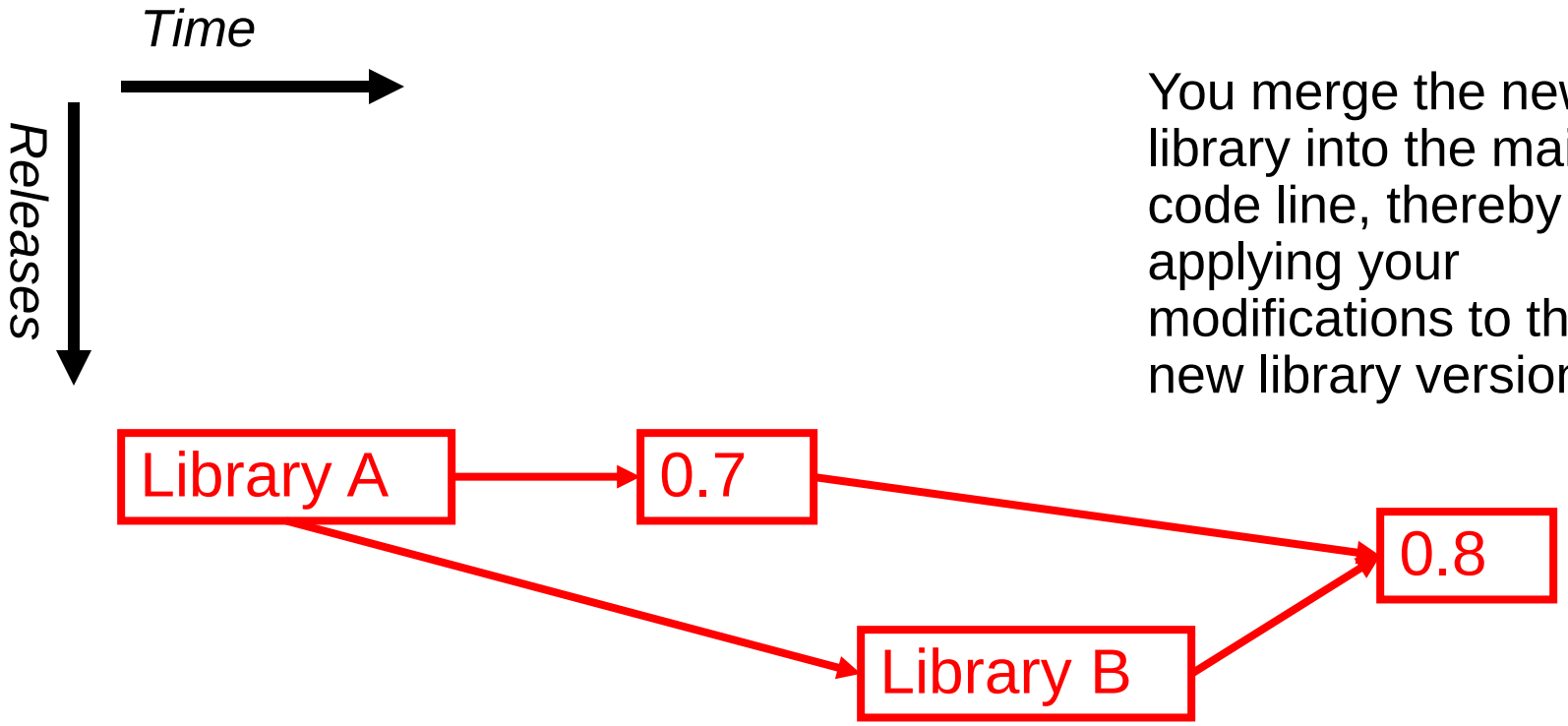
Scenario IV: Libraries



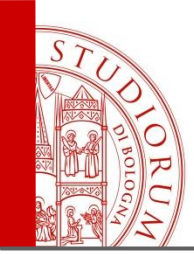
A new version of the library is released.

Logically this is a branch: library development has proceeded independently of your own development.

Scenario IV: Libraries



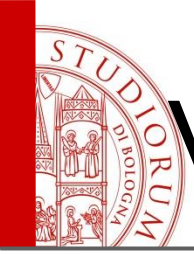
You merge the new library into the main code line, thereby applying your modifications to the new library version.



Home-made solution

Developers simply retain multiple copies of the different versions of the program, and label them appropriately

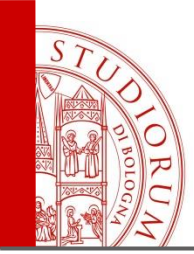
- Inefficient
  - many near-identical copies of the program have to be maintained
- It requires granting read-write-execute permissions to a set of developers
  - this adds the pressure of someone managing permissions so that the code base is not compromised



Version Control System (VCS)

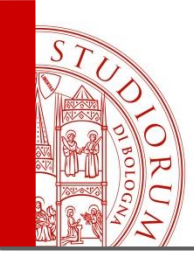
A VCS:

- Supports storing source code
- Provides a history of changes
- Provides a way to work in parallel on different aspects of the SW at hand
- Provides a way to work in parallel without interfering
- Provides a development model
- Increases productivity



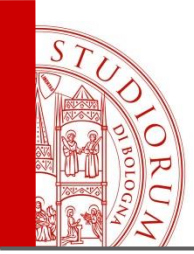
Centralized VCS: concepts

- Projects
- Repositories
- Working folders
- Revisions
- Branches
- Merging
- Conflicts



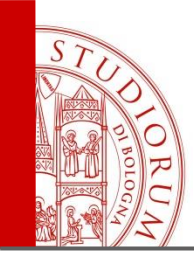
Projects

- A project is a set of files in version control
- Version control doesn't care what files
  - Not a build system
  - Or a test system
    - Though there are often hooks to these other systems
  - Just manages versions of a collection of files



Repository

- The repository is where files' current and historical data are stored
  - Typically, is on a remote server, on a reliable and secure machine
  - All users share the same repository
  - Sometimes called a depot



Working folder

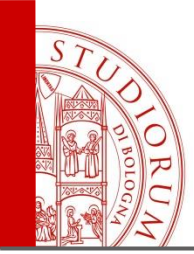
- The working copy is the local copy of files from a repository, at a specific time
  - All work done to the files in a repository is initially done on a working copy, hence the name
  - Every developer has one on her machine
  - Conceptually, it is a sandbox
    - An environment that isolates untested code changes and outright experimentation from the production environment

Working folder

- Copying the content of the repository to a working folder is called check out
- When the user is finished, changes can be committed to the repository, called check in

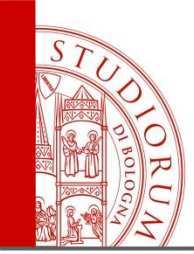
The workflow

- Copy the files from the repository into your working folder
- Modify the code into your working folder
- Update the repository from your working folder
  - First check that your code is correct
- Start all over again…



Revisions

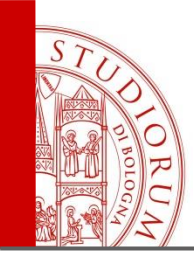
- Consider
  - Check out a file
  - Edit it
  - Check the file back in
- This creates a new version of the file
  - Usually increment minor version number
  - E.g., 1.5 → 1.6



Revisions (cont.)

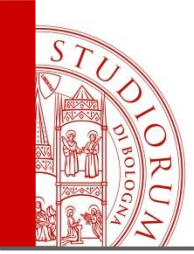
- Observation: Most edits are small

- For efficiency, don’t store entire new file
  - Store diff with previous version
  - Minimizes space
  - Makes check-in, check-out potentially slower
    - Must apply diffs from all previous versions to compute current file



Revisions (cont.)

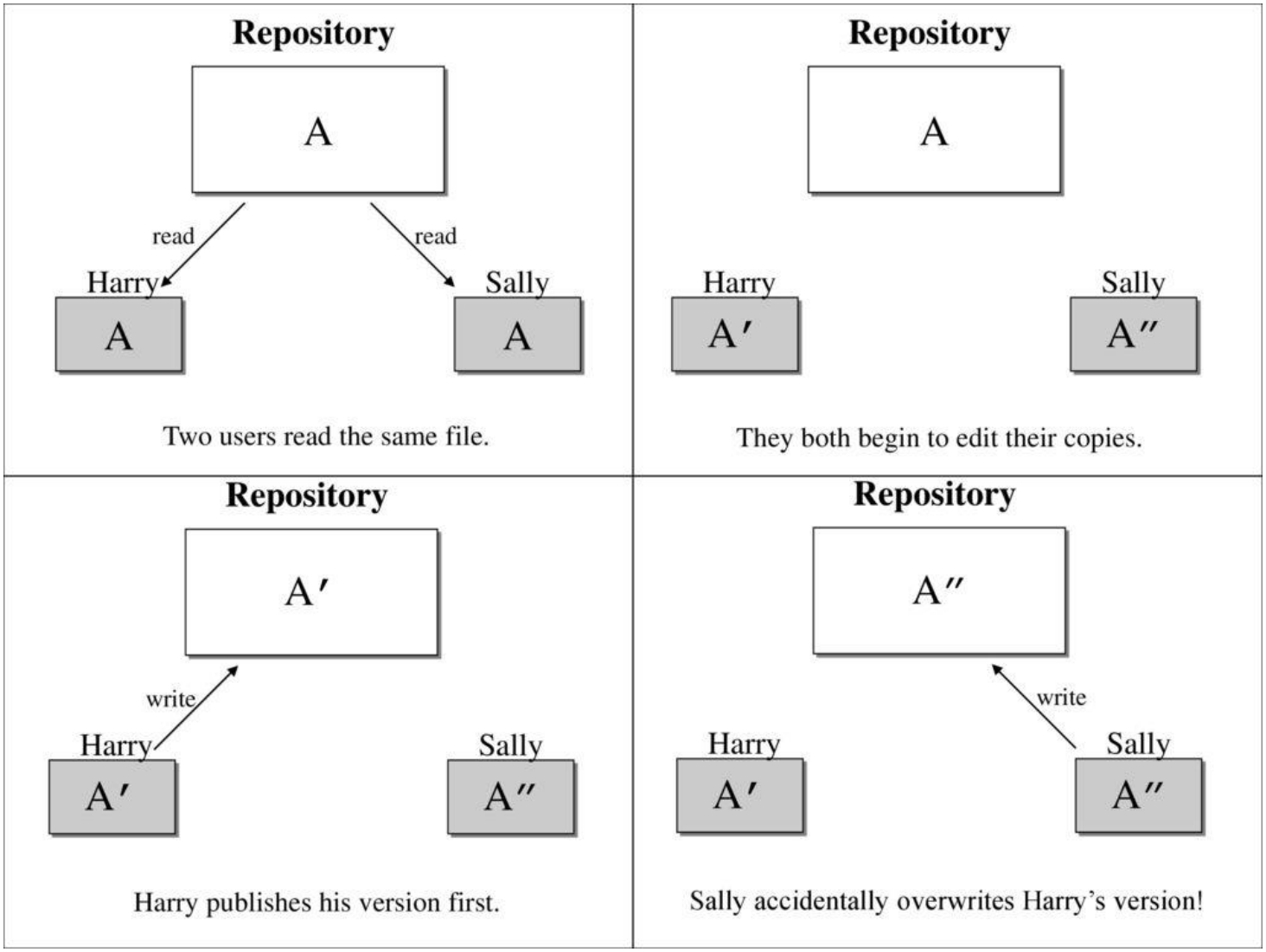
- With each revision, system stores
  - The diffs for that version
  - The new minor version number
  - Other metadata
    - Author
    - Time of check in
    - Log file message
    - Results of “smoke test” (a.k.a. “build acceptance test”)

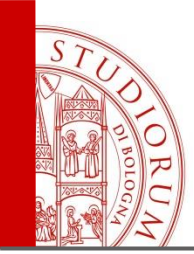


Branches

- A branch is just two revisions of a file
  - Two people check out 1.5
  - Check in 1.5.1
  - Check in 1.5.2
- Notes
  - Normally checking in does not create a branch
    - Changes merged into main code line
  - Must explicitly ask to create a branch

The problem to avoid

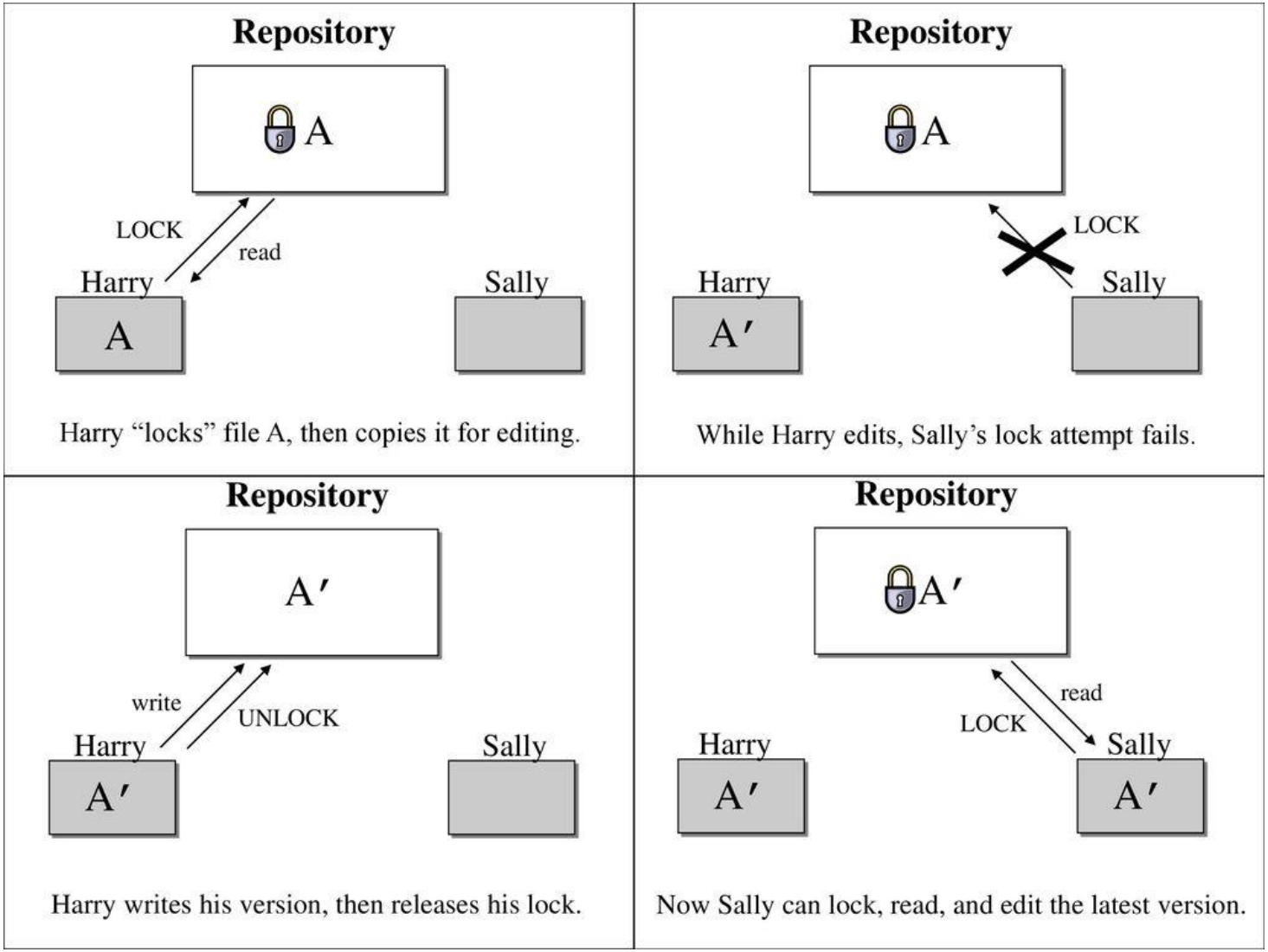


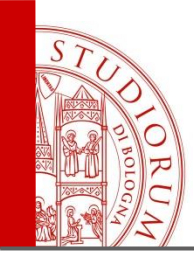


Lock-Modify-Unlock model

- The repository allows only one person to change a file at a time
- Every time someone wants to modify a file, it should first lock it
- If a file is locked, no other user can modify it
- When a user modifying a file has finished, it unlocks the file

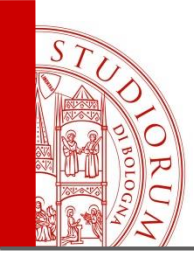
Lock-Modify-Unlock model





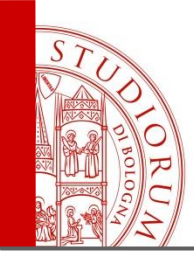
Lock-Modify-Unlock model

- Locking may cause administrative problems
  - Sometimes Harry will lock a file and then forget about it
  - Meanwhile, because Sally is still waiting to edit the file, her hands are tied
  - And then Harry goes on vacation
  - Now Sally has to get an administrator to release Harry's lock
  - The situation ends up causing a lot of unnecessary delay and wasted time



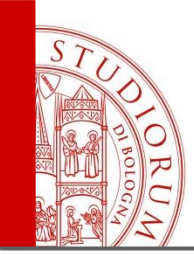
Lock-Modify-Unlock model

- Locking may cause unnecessary serialization
  - What if Harry is editing the beginning of a text file, and Sally simply wants to edit the end of the same file?
  - These changes don't overlap at all
  - They could easily edit the file simultaneously, and no great harm would come, assuming the changes were properly merged together
  - There's no need for them to take turns in this situation



Lock-Modify-Unlock model

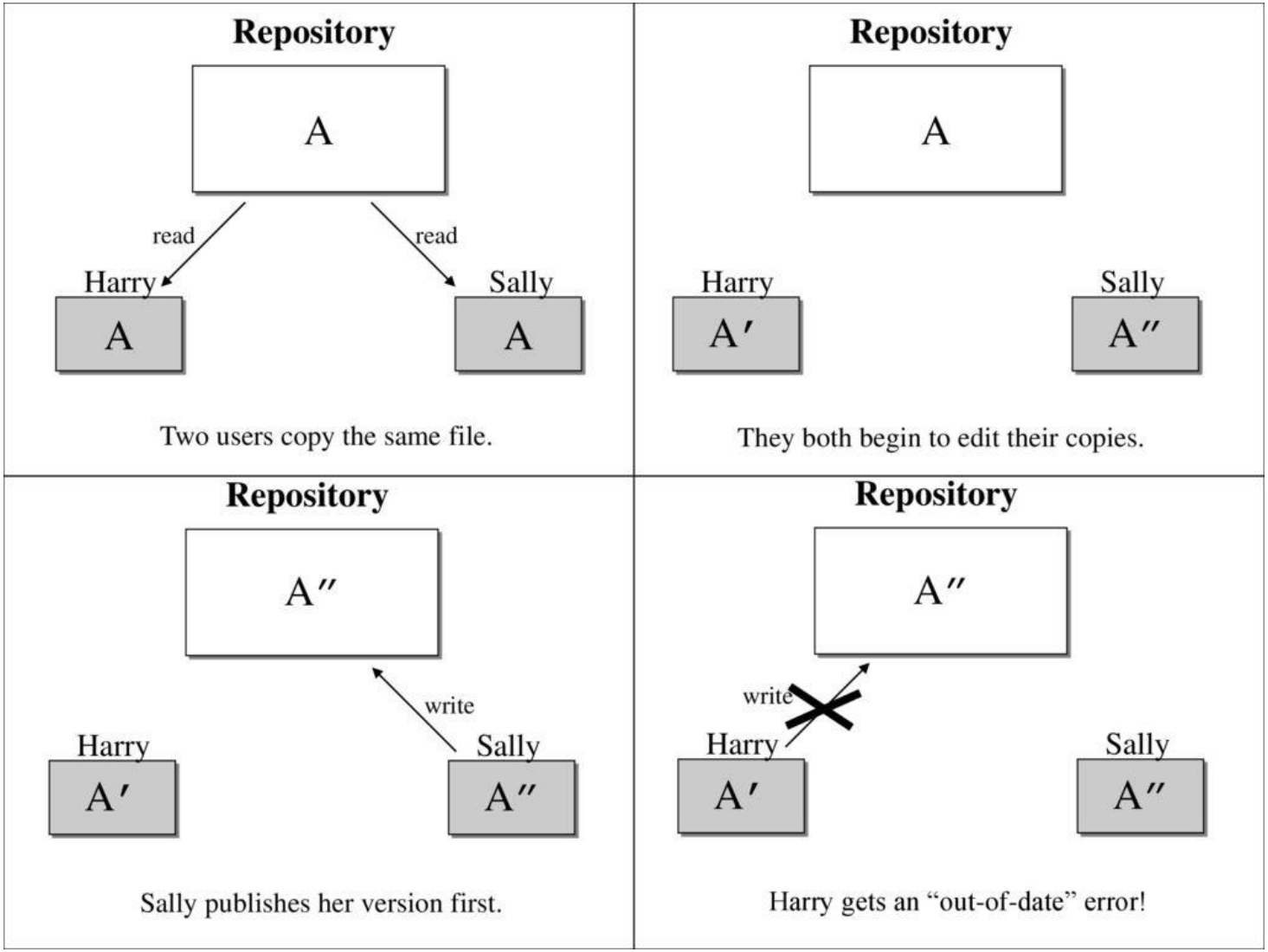
- Locking may create a false sense of security
  - Pretend that Harry locks and edits file A, while Sally simultaneously locks and edits file B
  - But suppose that A and B depend on one another, and the changes made to each are semantically incompatible
  - Suddenly A and B don't work together anymore
  - The locking system was powerless to prevent the problem
- How do you work offline?



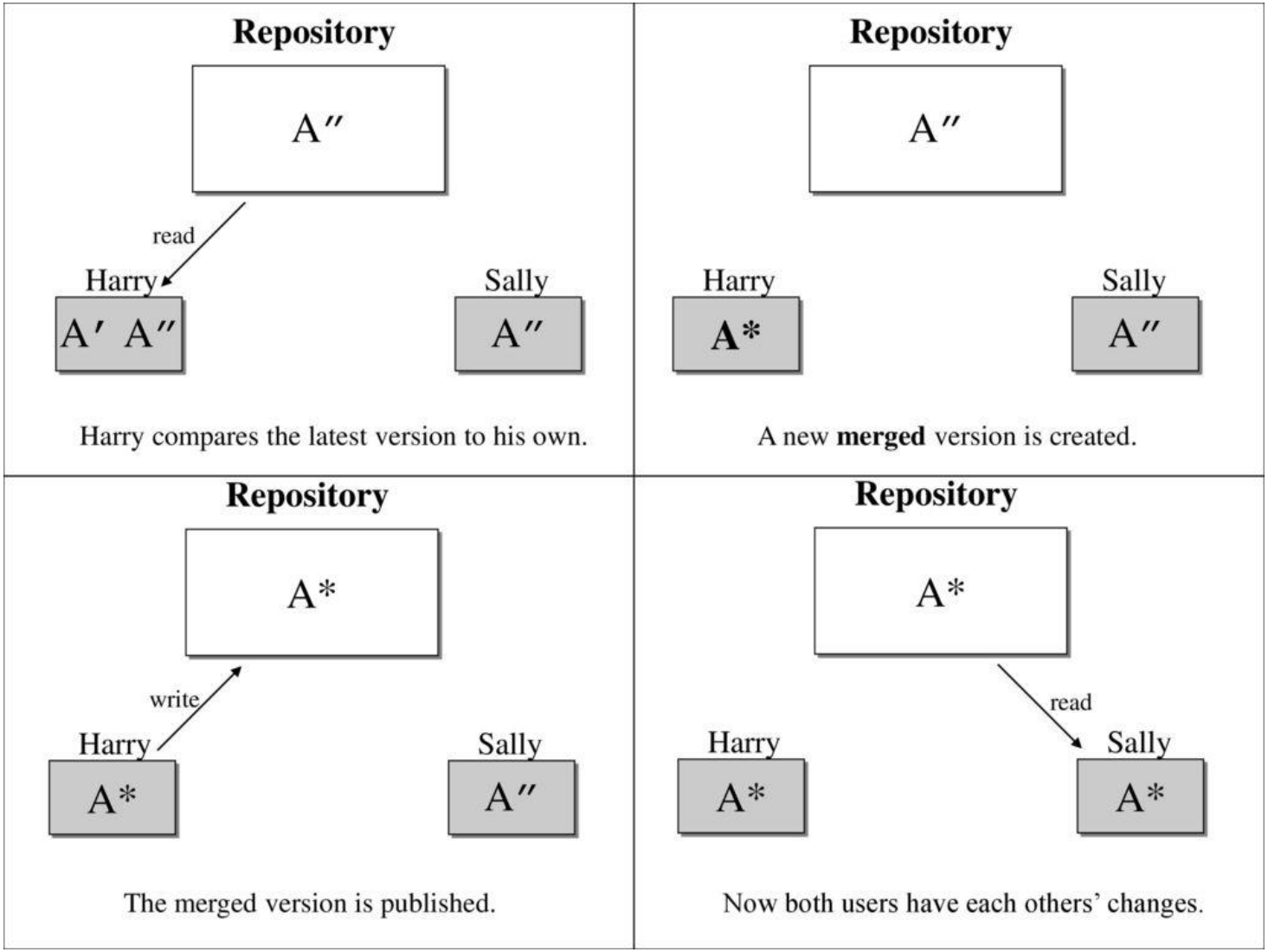
Copy-Modify-Merge model

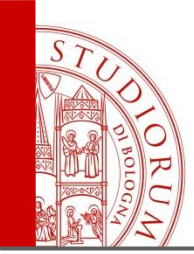
- No lock exists
- Whenever someone checks in a working copy, her edits are merged to those of other users

Copy-Modify-Merge model



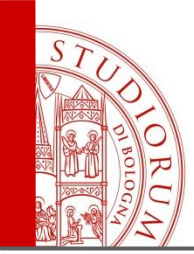
Copy-Modify-Merge model





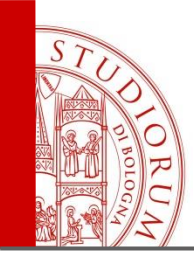
Merging

- Start with a file, say 1.5
- Bob makes changes A to 1.5
- Alice makes changes B to 1.5
- Assume Alice checks in first
  - Current revision is 1.6 = apply(B,1.5)



Merging (cont.)

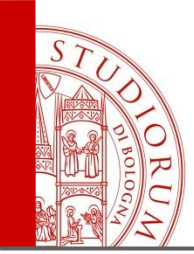
- Now Bob checks in
  - System notices that Bob checked out 1.5
  - But current version is 1.6
  - Bob has not made his changes in the current version!
- The system complains
  - Bob is told to *update* his local copy of the code



Merging (cont.)

- Bob does an update
  - This applies Alice's changes B to Bob's code
    - Remember Bob's code is apply(A,1.5)

- Two possible outcomes of an update
  - Success
  - Conflicts



Success

- Assume that

apply(A,apply(B,1.5))=apply(B,apply(A,1.5))

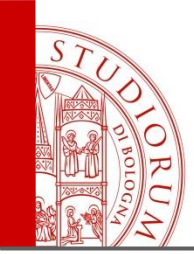
- Then then order of changes didn't matter
  - Same result whether Bob or Alice checks in first
  - The version control system is happy with this
- Bob can now check in his changes
  - Because apply(B,apply(A,1.6)) = apply(B,1.6)

Failure

- Assume

apply(A,apply(B,1.5))≠apply(B,apply(A,1.6))

- There is a *conflict*
  - The order of the changes matters
  - Version control will complain

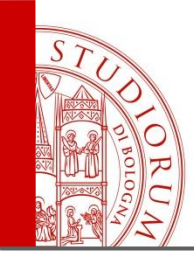


Conflicts

- Arise when two programmers edit the same piece of code
  - One change overwrites another

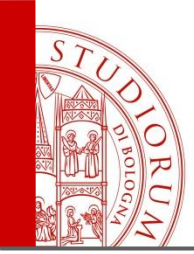
```
1.5:    a = b;
Alice:  a = b++;
Bob:    a = ++b;
```

The system doesn't know what should be done, and so complains of a conflict



Conflicts (cont.)

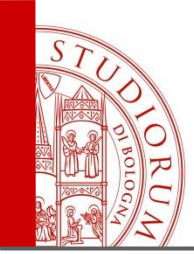
- System cannot apply changes when there are conflicts
  - Final result is not unique
  - Depends on order in which changes are applied
- Version control shows conflicts on update
  - Generally based on diff3
- Conflicts must be resolved by hand



Conflicts are syntactic

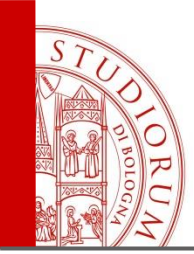
- Conflict detection is based on “nearness” of changes
  - Changes to the same line will conflict
  - Changes to different lines will likely not conflict

- Note: Lack of conflicts *does not* mean Alice’s and Bob’s changes work together



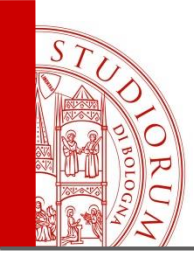
Example with no conflict

- Revision 1.5: `int f(int a, int b) { … }`
- Alice: `int f(int a, int b, int c) { … }`
  add argument to all calls to `f`
- Bob: add call `f(x,y)`
- Merged program
  - Has no conflicts
  - But will not even compile



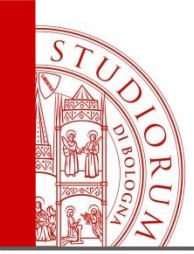
Don’t forget…

- Merging is syntactic
- Semantic errors may not create conflicts
  - But the code is still wrong
  - You are lucky if the code doesn’t compile
    - Worse if it does…



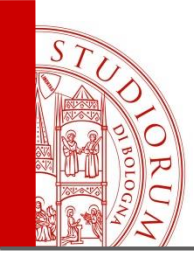
LMU vs. CMM

- CMM wins!
  - The copy-modify-merge model may sound a bit chaotic, but in practice, it runs extremely smoothly
  - Users can work in parallel, never waiting for one another
  - It turns out that most of their concurrent changes don't overlap at all; conflicts are infrequent
  - The amount of time it takes to resolve conflicts is far less than the time lost by a locking system
- LMU wins!
  - Unmergeable files (e.g., graphic images)



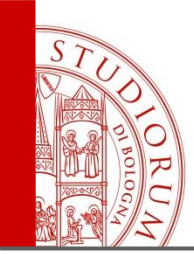
Centralized VCSs

- Concurrent Versions System (CVS)
  - 1986-2008
  - Only controls files, not directories, nor metadata
- Subversion (SVN)
  - 2000-
  - Apache project (2009-)



Moving to Distributed VCS

- Using CMM, the central repository is somehow losing its power
- Rather than a single, central repository on which clients synchronize, each peer's working copy of the codebase acts as a repository
- Distributed revision control conducts synchronization by exchanging patches (change-sets) from peer to peer

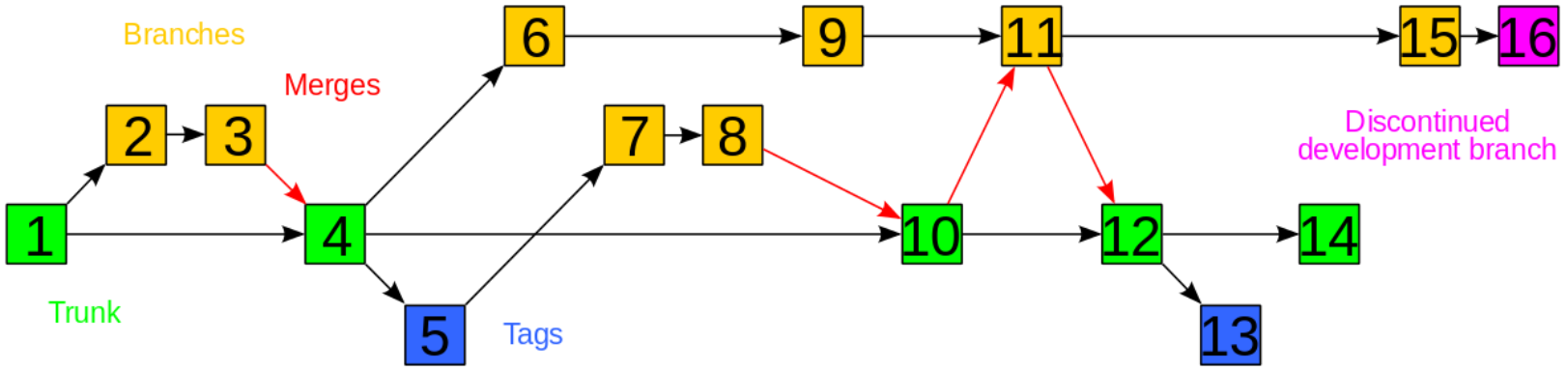


DVCS vs. CVCS

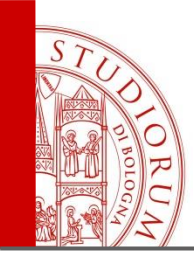
- No canonical, reference copy of the codebase exists, only working copies
  - However, you can create a central “official” repository
- Common operations (such as commits, viewing history, and reverting changes) are fast
  - No need to communicate with a central server
- Each working copy effectively functions as a remote backup of the codebase and of its change-history
  - More on this later…

The revision tree

Revisions are generally thought of as a line of development (the trunk) with branches off of this, forming a directed tree, visualized as one or more parallel lines of development (the “mainlines” of the branches) branching off a trunk

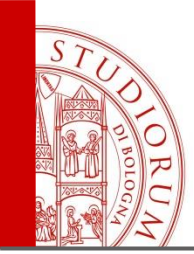


In reality, the structure is more complicated, forming a directed acyclic graph, but for many purposes “tree with merges” is an adequate approximation



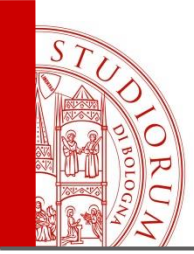
Distributed VCS: concepts

- Trunk
  - The unique line of development (also called Baseline, Mainline, or Master)
- Branch
- Tag
- Push/Pull



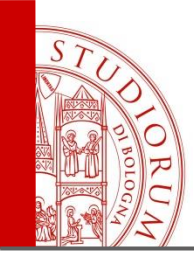
Branch

- A set of files under version control may be branched at a point in time so that, from that time forward, two copies of those files may develop at different speeds or in different ways independently of each other
- It is a complete copy of the codebase (including its history)
- When updates are consolidated, they can be merged to the main trunk



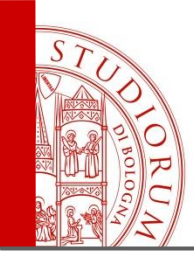
Tag

- A tag (or label) refers
  to an important snapshot in time,
  consistent across many files
- These files at that point may all be tagged
  with a user-friendly, meaningful name
  or revision number
- Tag = Release



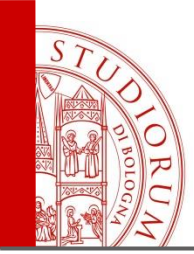
Push/Pull

- Copy revisions from one repository into another
- Push is initiated by the source
- Pull is initiated by the receiving repository
  - Usually, the contributor issues a pull request
  - The maintainer has to merge the pull request



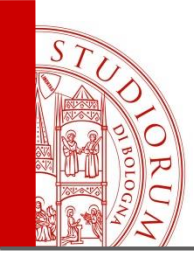
The workflow

- For others to see your changes, 4 things must happen:
  - You commit
  - You push
  - They pull
  - They update
- The commit and update commands only move changes between the working copy and the local repository
- By contrast, the push and pull commands move changes between the local repository and other repositories



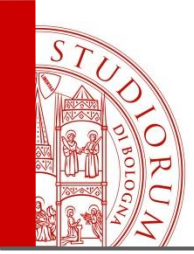
What am I pushing/pulling?

- Distributed version control focuses on sharing changes
  - every change has a GUID
- Push/Pull actions use only change sets, not actual files
- Every change is easy to track (thanks to its GUID)



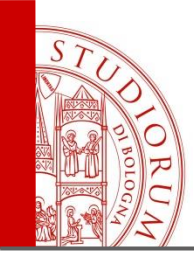
Best practices

- Use a descriptive commit message
  - This indicates the purpose of the change
- Make each commit a logical unit
  - This makes it easier to locate the changes related to some particular feature or bug fix
- Avoid indiscriminate commits
  - You may have changes you do not intend to make permanent



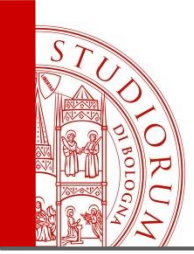
Best practices (cont.)

- Incorporate others' changes frequently
  - Conflicts will be rather infrequent
- Share your changes frequently
  - Same as above
- Coordinate with your co-workers
  - Again…
- Don't commit generated files



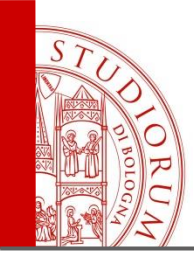
DVCS: pros

- Everyone has a local sandbox
- It works offline
- It's fast
  - Diffs, commits, and reverts are all done locally
- Branching and merging is easy
  - The GUIDs make it easy to automatically combine changes and avoid duplicates
- Less management



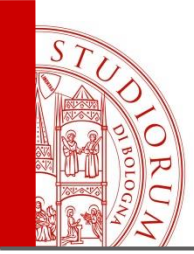
DVCS: cons

- You still need a backup
  - Other users might not have accepted your changes…
- There's not really a "latest version"
  - A central location helps clarify what the latest "stable" release is
- There aren't really revision numbers
  - GUIDs are not release numbers
  - But you can always tag!



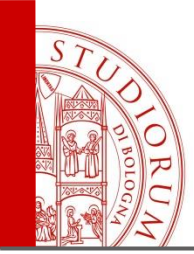
Distributed VCSs

- BitKeeper
  - 2000-2005 (free)
    - In 2002 used for Linux kernel development
    - Flamewars (are we using proprietary SW for Linux?!?!)
  - 2005-2016 (commmercial)
  - 2016- (open-source)



Distributed VCSs

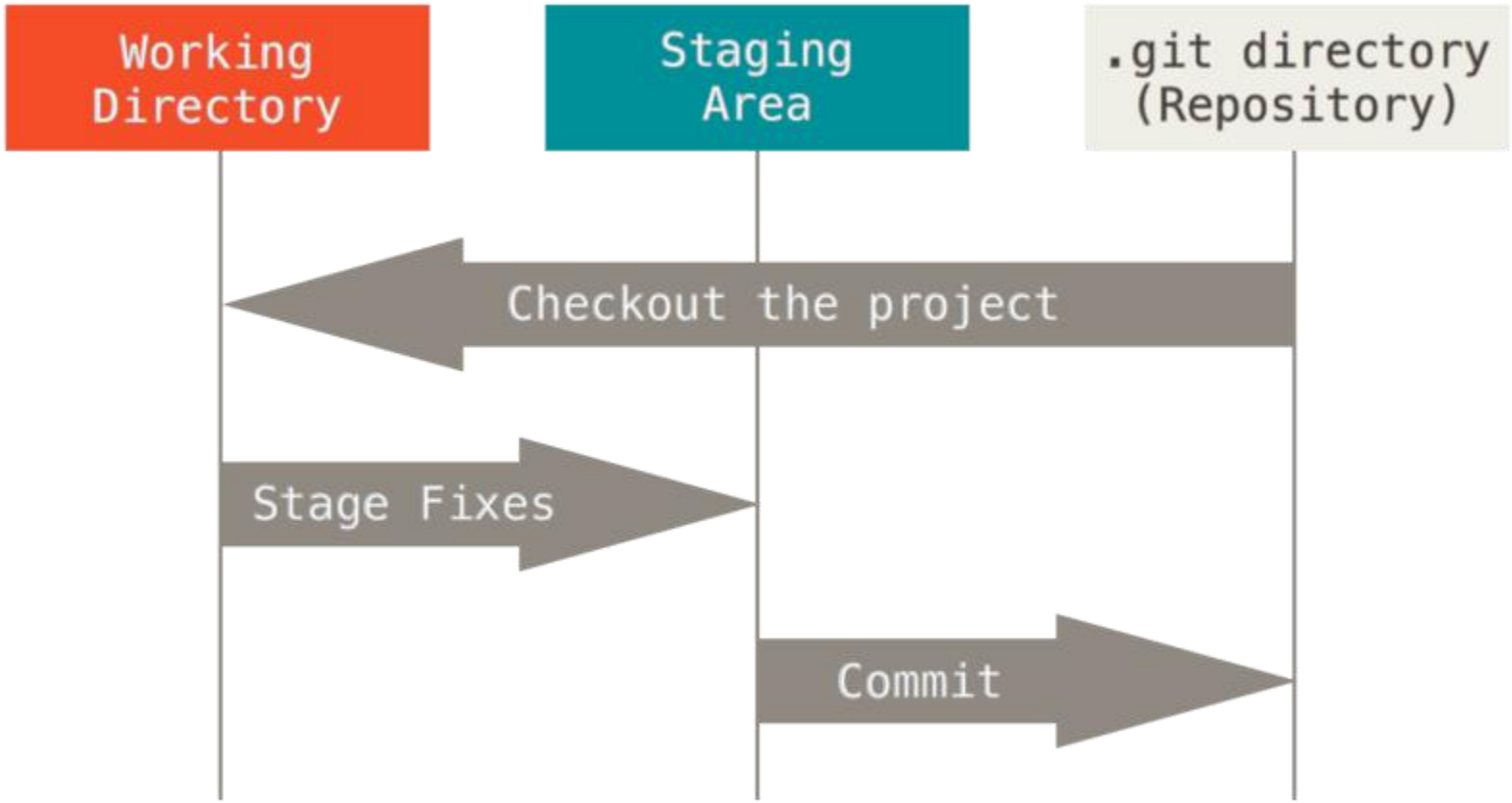
- Mercurial
  - 2005-
  - Counteract BitKeeper being no longer free
- Git
  - 2005-
  - Created by Linus Torvalds for Linux kernel

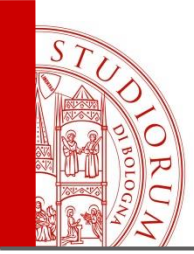


Git: concepts

- Directory
  - It's the main (centralized) repository
- Working directory
  - A local copy of the repository
- Staging area
  - A file (index) specifying which modified files should be saved (committed) to the repository

Git: concepts

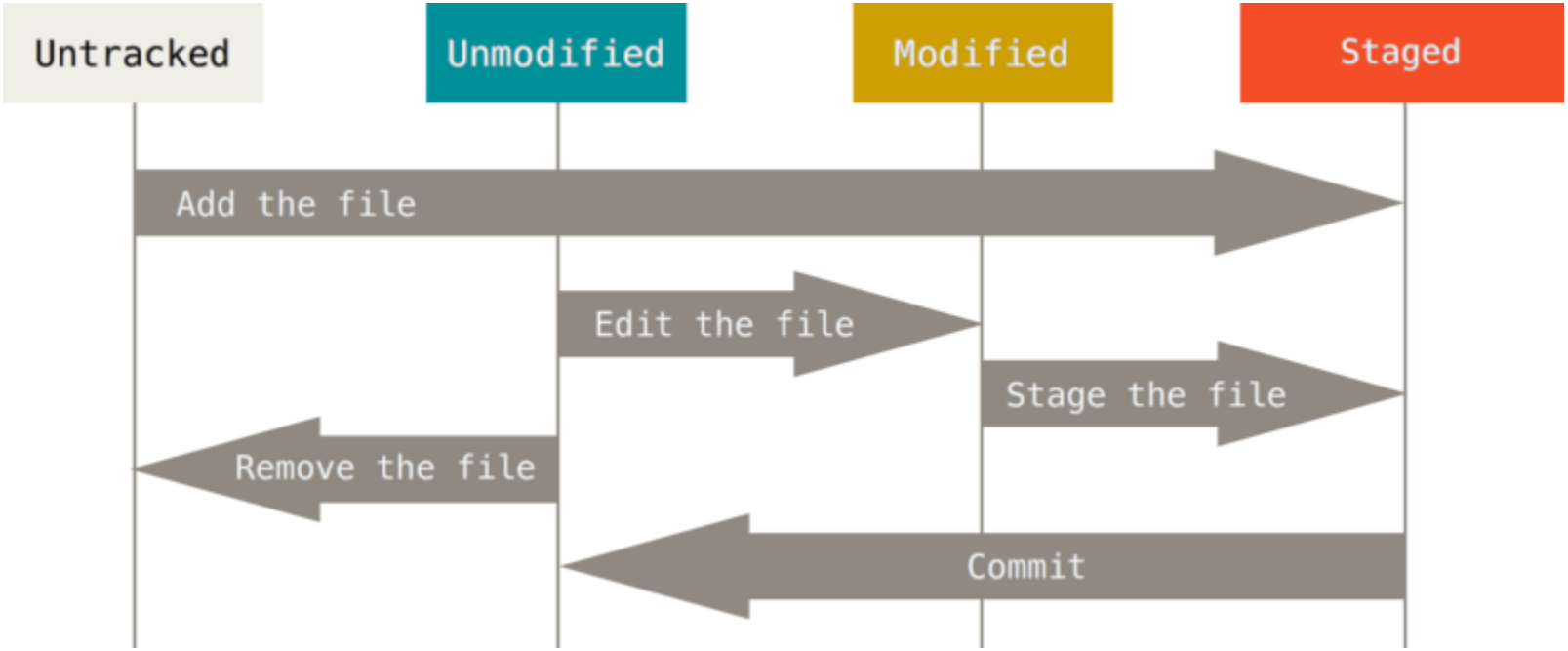


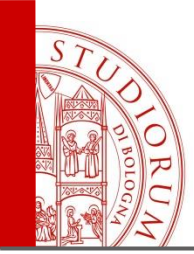


Git: status of a file

- Committed
  - The file is saved in the (local) repository
- Modified
  - It has been modified,
    but have not been committed yet
- Staged
  - It has been marked to be committed next

Git: status of a file



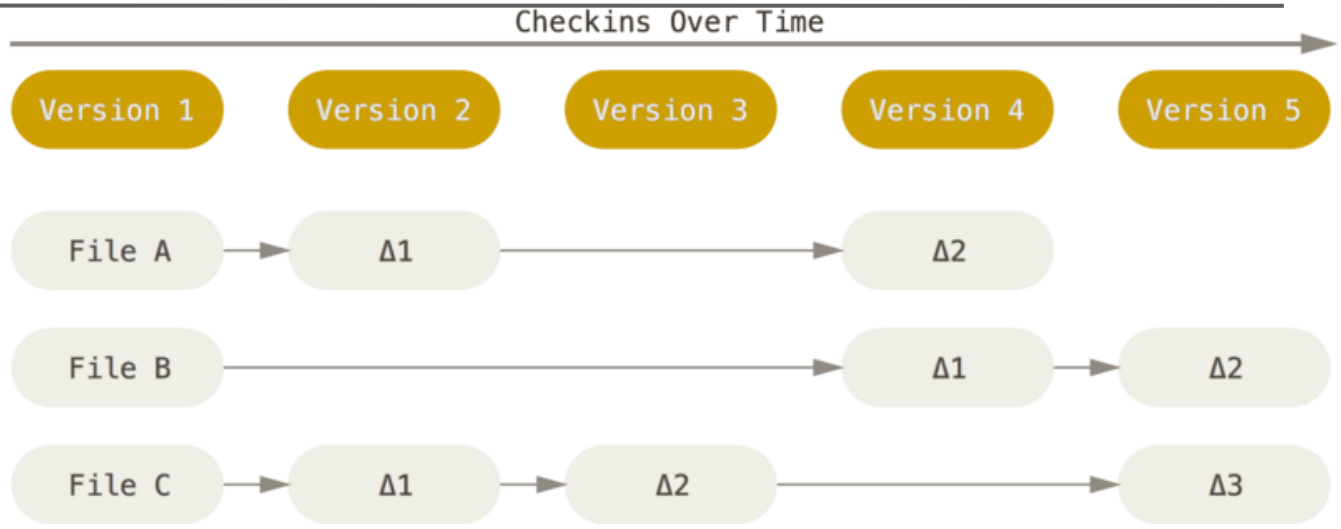


Git: the workflow

- Checkout the project
  - From the Git directory
    into your working directory
- Modify files
  - Such edits remain within the working directory
- Stage files
  - Selectively stage just those changes
    you want to be part of your next commit
- Commit
  - Store snapshot files permanently

Git stores snapshots, not differences

- Other VCSs



- Git